Embedded.com

Watching the Watchdog

By Jack G. Ganssle, Embedded Systems Programming
Feb 20, 2003 (10:30 AM)
URL: http://www.embedded.com/story/OEG20030220S0037

Although good watchdog hardware is key, it's the software that really makes it work. Jack explains how to keep an eye on your dog.

Since watchdog timers (WDTs) are our last line of defense when the code collapses, they had better be foolproof. Let's take a look at how they work in multitasking systems.

Tasking turns a linear bit of software into a multidimensional mix of tasks competing for processor time. Each runs more or less independently of the others, which means each can crash on its own, without bringing the entire system to its knees.

You can learn a lot about a system's design just by observing its operation. Consider a simple instrument with a display and various buttons. Press a button and hold it down; if the display continues to update, odds are the system multitasks. Yet, in the same system, a software crash might go undetected by conventional watchdog strategies. If the display or keyboard tasks die, the mainline code or a watchdog task may continue to run.

Any system that uses an interrupt service routine (ISR) or a special task to tickle the watchdog but doesn't examine the health of all other tasks is not robust. You've got to weave the watchdog's kick into the fabric of the system's tasks, which is, happily, easier than it sounds.

First, build a watchdog task. It's the only part of the software allowed to tickle the EDT hardware. If your system has a memory management unit, mask off all I/O accesses to the WDT except those from this task. With I/O accesses, rogue code traps on an errant attempt to output to the watchdog.

Next, create a data structure that has one entry per task, with each entry being just an integer. When a task starts, it increments its entry in this structure. Tasks that only start once and stay active forever can increment the appropriate value each time they run through their outer loops.

Increment the data atomically in a way that cannot be interrupted with the data half-changed. **++TASK[i]** (if **TASK** is an integer array) on an 8-bit CPU might not be atomic, though it's almost certainly okay on a 16- or 32-bitter. The safest way to both encapsulate and ensure atomic access to the data structure is to hide it behind another task. Use a semaphore to eliminate concurrent shared accesses. Send increment messages to the task, using the RTOS's messaging resources.

As the program runs, the number of counts for each task advances. Infrequently but at regular intervals the watchdog task runs: once a second, once a millisecond—it's a function of paranoia and the implications of a failure.

The watchdog task scans the structure, checking that the count stored for each task is reasonable. One that runs often should have a high count; one that executes infrequently will produce a smaller value. Part of the trick is determining what's reasonable for each task. If the counts are unreasonable, halt and let the watchdog timeout. If everything is okay, set all of the counts to zero and exit. Why is this robust? Obviously, the watchdog monitors every task in the system. But it's also impossible for code that's running amok to stumble into the WDT task and errantly tickle the dog; by zeroing the array, we guarantee it's in a "bad" state.

Add to the above a windowing WDT circuit, like the one described last month ("Li'l Bow-Wow," February 2003, p. 31). That way, even if the code does accidentally send the right tickle codes, it will almost certainly violate the timing window, causing a timeout.

Michael Barr, this magazine's editor in chief, likes to run the watchdog task at a high priority to be sure he can record whatever task ran amok before nuking the system. Keeping the priority high also ensures the task runs pretty much when it's supposed to, so we always tickle within the timing set up by the watchdog's window.

Quantifying "reasonable"

How do we decide what's a reasonable count for each task? Sometimes, we can figure it out analytically. If the WDT task runs once a second, and one of the monitored tasks starts every 50ms, then a count of around 20 is reasonable.

Determining count times for other activities can be more difficult. What about a task that responds to asynchronous inputs from other computers, such as data packets that come irregularly? Or a periodic event that drives a low-priority task, which can be suspended for long intervals by higher-priority tasks.

The solution is to broaden the data structure that maintains count information. Add **min** and **max** fields to each entry. Each task must run at least **min**, but no more than **max** times. Now redesign the watchdog task to run in one of two modes. The first is the one already described and is used during normal system operation.

The second mode is a debug environment enabled by a compile-time switch that collects **min** and **max** data. Each time the WDT task runs it looks at the incremented counts and updates the **min** and **max** values as needed. It still tickles the watchdog each time it executes.

Run the product's full test suite with this mode enabled for as long as it takes to get a profile of the **min/max** values. When you're satisfied the tests are representative of the system's real operation, manually examine the data and adjust the parameters as seems necessary to give adequate margins to the data.

It's a chore, but taking this step yields a great watchdog and a deep look into your system's timing. And timing is every bit as important as procedural issues, though it often goes ignored until a nagging problem turns into an unacceptable symptom. This watchdog scheme forces you to think in the time domain and, by its nature, profiles—crudely—your system's timing.

There's one more kink. Some tasks run so infrequently or erratically that any sort of automated profiling will fail. A watchdog that runs once a second will miss tasks that start only hourly. It's not unreasonable to exclude these from watchdog monitoring. Or we can add a bit of complexity to the code to initiate a watchdog timeout if, say, the slow tasks don't start even after some number of hours elapse.

Beyond watchdogs

I'm troubled by the fan failure I referenced in my January column ("Born to Fail," p. 32). A vent fan over a kitchen stove ran amok, ignoring user input until the owner cycled power. Since that column ran, two readers informed me about identical situations with their vent fans, all apparently made by the same vendor.

The fan failures didn't make the news or hurt anyone so why worry? Because headlines and physical harm aren't the only ways to measure damage. If we can't build reliable fan controllers, what hope is there for mission-critical applications?

I don't know what went wrong with those fan controllers, and I have no idea if a WDT is part of the system. Either way, such failures are avoidable and unacceptable. And watchdogs are only part of the picture. A WDT tells us the code is running. A windowing WDT tells us it's running with mostly correct timing. But no watchdog directly flags software executing with corrupt data structures. Why does a data structure become corrupt? Bugs, for one thing. Conditions the designers didn't anticipate also create problems.

So we need more self-defense than watchdogs provide. Safety critical apps, where the cost of a failure is frighteningly high, should definitely include integrity checks on the data. Low-threat equipment like the vent fan can and should have at least a minimal amount of code for trapping possible failure conditions.

Some argue it makes no sense to "waste" time writing defensive code for a dumb fan application. Yet the simpler the system, the easier and quicker it is to plug in a bit of code to look for program and data errors. Very simple systems tend to translate inputs to outputs. Their primary data structures are the I/O ports. Often, several unrelated output bits get multiplexed to a single port. To change one bit means either reading the port's current status or maintaining a copy of the port in RAM. Both approaches are problematic.

Since computers are deterministic, it's reasonable to expect that, in the absence of bugs, they'll produce correct results all the time. It's apparently safe, therefore, to read a port's current status, **AND** off the unwanted bits, **OR** in new ones, and output the result. This is a state machine; the outputs evolve over time to deal with changing inputs. But the process works only if the state machine never incorrectly flips a bit. Unfortunately, output ports are connected to the hostile environment of the real world. It's entirely possible that a bit of energy from starting the fan's highly inductive motor will alter the port's setting. I've seen this happen many times.

So maybe it's more reliable to maintain a memory image of the port. The downside is that a program bug might corrupt the image. Most of the time they're stored as global variables, so any sloppy code can accidentally trash the location. Encapsulation solves that problem, but not the one of a wandering pointer walking over the data, or of a latent reentrancy issue corrupting things. You might argue that writing correct code means we shouldn't worry about a location changing, but we added a WDT in part to deal with bugs. Similar concerns about our data are warranted.

Jack G. Ganssle is a lecturer and consultant on embedded development issues. He conducts seminars on embedded systems and helps companies with their embedded challenges. Contact him at jack@ganssle.com.